Concurrent Processes

- What is Parallel Processing?
- Typical Multiprocessing Configurations
- Process Synchronization Software
- Process Cooperation
- Concurrent Programming
- Ada

Single Processor Configurations

Multiple Process Synchronization

Multiple Processor Programming
What is Parallel Processing?

• Parallel processing (multiprocessing) -- 2+ processors operate in unison.
  – 2+ CPUs are executing instructions simultaneously.
  – Each CPU can have a process in RUNNING state at same time.

• Processor Manager has to coordinate activity of each processor and synchronize interaction among CPUs.

• Synchronization is key to system’s success because many things can go wrong in a multiprocessing system.
Development of Parallel Processing

• Major forces behind the development of multiprocessing:
  – Enhance throughput
  – Increase computing power.

• Primary benefits:
  – increased reliability due to availability of 1+ CPU
  – faster processing because instructions can be processed in parallel, two or more at a time.

• Major challenges:
  – How to connect the processors into configurations
  – How to orchestrate their interaction
Typical Multiprocessing Configurations

- Master/slave
- Loosely coupled
- Symmetric
Master/Slave Multiprocessing Configuration

- Asymmetric configuration.
- Single-processor system with additional “slave” processors.
- Each slave, all files, all devices, and memory managed by primary “master” processor.
- Master processor maintains status of all processes in system, performs storage management activities, schedules work for the other processors, and executes all control programs.
Pros & Cons of Master/Slaves

• The primary advantage is its simplicity.

• Reliability is no higher than for a single processor system because if master processor fails, entire system fails.

• Can lead to poor use of resources because if a slave processor becomes free while master is busy, slave must wait until the master can assign more work to it.

• Increases number of interrupts because all slaves must interrupt master every time they need OS intervention (e.g., I/O requests).
Loosely Coupled Multiprocessing Configuration

- Features several complete computer systems, each with own memory, I/O devices, CPU, & OS.
- Each processor controls own resources, maintains own commands & I/O management tables.
- Each processor can communicate and cooperate with the others.
- Each processor must have “global” tables indicating jobs each processor has been allocated.
Loosely Coupled - 2

- To keep system well-balanced & ensure best use of resources, job scheduling is based on several requirements and policies.
  - E.g., new jobs might be assigned to the processor with lightest load or best combination of output devices available.

- System isn’t prone to catastrophic system failures because even when a single processor fails, others can continue to work independently from it.

- Can be difficult to detect when a processor has failed.
Symmetric Multiprocessing Configuration

- Processor scheduling is decentralized.
- A single copy of OS & a global table listing each process and its status is stored in a common area of memory so every processor has access to it.
- Each processor uses same scheduling algorithm to select which process it will run next.
Advantages of Symmetric over Loosely Coupled Configurations

• More reliable.
• Uses resources effectively.
• Can balance loads well.
• Can degrade gracefully in the event of a failure.

• Most difficult to implement because processes must be well synchronized to avoid problems of races and deadlocks.
Process Synchronization Software

• Success of process synchronization hinges on capability of OS to make a resource unavailable to other processes while it’s being used by one of them.
  – E.g., I/O devices, a location in storage, or a data file.

• In essence, used resource must be locked away from other processes until it is released.

• Only when it is released is a waiting process allowed to use resource. A mistake could leave a job waiting indefinitely.
Synchronization Mechanisms

• Common element in all synchronization schemes is to allow a process to finish work on a **critical region** of program before other processes have access to it.
  – Applicable both to multiprocessors and to 2+ processes in a single-processor (time-shared) processing system.

• Called a critical region because its execution must be handled as a unit.
Lock-and-Key Synchronization

• Process first checks if key is available
• If it is available, process must pick it up and put it in lock to make it unavailable to all other processes.

• For this scheme to work both actions must be performed in a single machine cycle.

• Several locking mechanisms have been developed including test-and-set, WAIT and SIGNAL, and semaphores.
Test-And-Set (TS) Locking

- **Test-and-set** is a single indivisible machine instruction (TS).
- In a single machine cycle it tests to see if key is available and, if it is, sets it to “unavailable.”
- Actual key is a single bit in a storage location that can contain a zero (if it’s free) or a one (if busy).
- Simple procedure to implement.
- Works well for a small number of processes.
Problems with Test-And-Set

• When many processes are waiting to enter a critical region, *starvation* could occur because processes gain access in an arbitrary fashion.
  – Unless a first-come first-served policy were set up, some processes could be favored over others.

• Waiting processes remain in unproductive, resource-consuming wait loops (*busy waiting*).
  – Consumes valuable processor time.
  – Relies on the competing processes to test key.
WAIT and SIGNAL Locking

- Modification of test-and-set.

- Adds 2 new operations, which are mutually exclusive and become part of the process scheduler’s set of operations
  - WAIT
  - SIGNAL

- Operations WAIT and SIGNAL frees processes from “busy waiting” dilemma and returns control to OS which can then run other jobs while waiting processes are idle.
WAIT

• Activated when process encounters a busy condition code.

• Sets process control block (PCB) to the blocked state

• Links it to the queue of processes waiting to enter this particular critical region.

• Process Scheduler then selects another process for execution.
- Activated when a process exits the critical region and the condition code is set to “free.”

- Checks queue of processes waiting to enter this critical region and selects one, setting it to the READY state.

- Process Scheduler selects this process for running.
Semaphores

• **Semaphore** -- nonnegative integer variable used as a flag.
• Signals if & when a resource is free & can be used by a process.
• Most well-known semaphores are signaling devices used by railroads to indicate if a section of track is clear.
• Dijkstra (1965) -- 2 operations to operate semaphore to overcome the process synchronization problem.
  – P stands for the Dutch word *proberen* (to test)
  – V stands for *verhogen* (to increment)
P (Test) and V (Increment)

- If we let $s$ be a semaphore variable, then the V operation on $s$ is simply to increment $s$ by 1.
  \[ V(s): s := s + 1 \]

- Operation P on $s$ is to test value of $s$ and, if it’s not zero, to decrement it by one.
  \[ P(s): \text{If } s > 0 \text{ then } s := s - 1 \]

- P and V are executed by OS in response to calls issued by any one process naming a semaphore as parameter.
MUTual EXclusion (Mutex)

- P and V operations on semaphore $s$ enforce concept of mutual exclusion, which is necessary to avoid having 2 operations attempt to execute at same time.

- Called mutex (MUTual EXclusion)

  \[
  \begin{align*}
  \text{P(mutex)}: & \quad \text{if } mutex > 0 \text{ then } mutex: = mutex - 1 \\
  \text{V(mutex)}: & \quad \text{mutex:} = mutex + 1
  \end{align*}
  \]
Process Cooperation

- Occasions when several processes work directly together to complete a common task.

- Two famous examples are problems of “producers and consumers” and “readers and writers.”

- Each case requires both mutual exclusion and synchronization, and they are implemented by using semaphores.
Producers and Consumers: One Process Produces Some Data That Another Process Consumes Later.
Producers and Consumers - 2

- Because buffer holds finite amount of data, synchronization process must delay producer from generating more data when buffer is full.

- Delay consumer from retrieving data when buffer is empty.

- This task can be implemented by 3 semaphores:
  - Indicate number of full positions in buffer.
  - Indicate number of empty positions in buffer.
  - Mutex, will ensure mutual exclusion between processes
# Definitions of Producer & Consumer Processes

<table>
<thead>
<tr>
<th>Producer</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>produce data</td>
<td>P (full)</td>
</tr>
<tr>
<td>P (empty)</td>
<td>P (mutex)</td>
</tr>
<tr>
<td>P (mutex)</td>
<td>read data from buffer</td>
</tr>
<tr>
<td>write data into buffer</td>
<td>V (mutex)</td>
</tr>
<tr>
<td>V (mutex)</td>
<td>V (empty)</td>
</tr>
<tr>
<td>V (full)</td>
<td>consume data</td>
</tr>
</tbody>
</table>
Definitions of Variables and Functions Used in Producers and Consumers

Given: Full, Empty, Mutex defined as semaphores

n: maximum number of positions in the buffer

V (x): \( x = x + 1 \) (x is any variable defined as a semaphore)

P (x): if \( x > 0 \) then \( x = x - 1 \)

mutex = 1 means the process is allowed to enter critical region
Producers and Consumers Algorithm

empty := n
full := 0
mutex := 1
COBEGIN
    repeat until no more data PRODUCER
    repeat until buffer is empty CONSUMER
COEND
Readers and Writers

• **Readers and writers** -- arises when 2 types of processes need to access a shared resource such as a file or database.
  – E.g., airline reservations systems.

• Two solutions using P and V operations:
  1. Give priority to readers over writers so readers are kept waiting only if a writer is actually modifying the data.

• However, this policy results in writer starvation if there is a continuous stream of readers.
Reader & Writer Solutions
Using P and V Operations

2. Give priority to the writers.
   • In this case, as soon as a writer arrives, any readers that are already active are allowed to finish processing, but all additional readers are put on hold until the writer is done.
   • Obviously this policy results in reader starvation if a continuous stream of writers is present
State of System Summarized By 4 Counters

- Number of readers who have *requested* a resource and haven’t yet released it (R1=0);
- Number of readers who are *using* a resource and haven’t yet released it (R2=0);
- Number of writers who have *requested* a resource and haven’t yet released it (W1=0);
- Number of writers who are *using* a resource and haven’t yet released it (W2=0).

- Implemented using 2 semaphores to ensure mutual exclusion between readers and writers.
Concurrent Programming

- **Concurrent processing system** -- one job uses several processors to execute sets of instructions in parallel.
  - Requires a programming language and a computer system that can support this type of construct.
- Increases computation speed.
- Increases complexity of programming language and hardware (machinery & communication among machines).
- Reduces complexity of working with array operations within loops, of performing matrix multiplication, of conducting parallel searches in databases, and of sorting or merging files.
Explicit & Implicit Parallelism

- **Explicit parallelism** -- programmer **must** explicitly state which instructions can be executed in parallel.

- **Implicit parallelism** -- automatic detection by compiler of instructions that can be performed in parallel.
Ada

- In early 1970s DoD commissioned a programming language that could perform concurrent processing.
- Named after Augusta Ada Byron (1815-1852), a skilled mathematician & world’s first programmer for work on Analytical Engine.
- Designed to be modular so several programmers can work on sections of a large project independently of one another.
  - Specification part, which has all information that must be visible to other units (argument list)
  - Body part made up of implementation details that don’t need to be visible to other units.
Terminology

- Ada
- busy waiting
- COBEGIN
- COEND
- concurrent processing
- concurrent programming
- critical region
- embedded computer systems
- explicit parallelism
- implicit parallelism
- loosely coupled configuration
- master/slave configuration
- multiprocessing
- mutex
- P
- parallel processing
- process synchronization
- producers and consumers
- readers and writers
- semaphore
- symmetric configuration
- test-and-set
- V
- WAIT and SIGNAL